

Entwurf und Implementierung paralleler Programme

Prof. Dr. Rolf Hennicker

17.06.2009

Kapitel 5

Monitore und

Synchronisationsbedingungen

5.1 Synchronisationsbedingungen

Bisherige Verwendung von Monitoren:

Verkapselung von Daten, Zugriffoperationen unter wechselseitigem Ausschluss.

Jetzt zusätzlich:

Synchronisationsbedingungen zur Prozesskooperation.

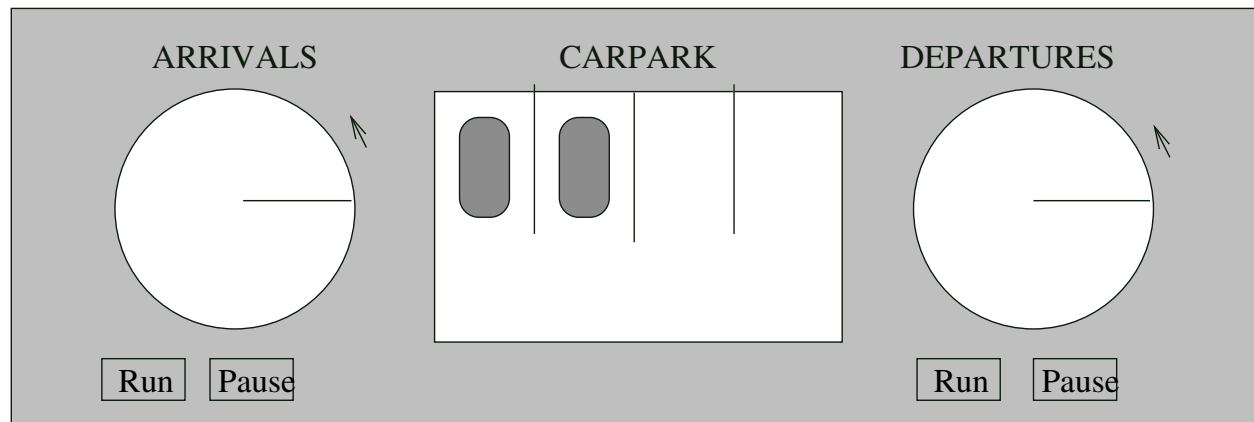
Idee:

Bestimmte Monitoroperationen dürfen erst dann ausgeführt werden, wenn eine bestimmte Bedingung (abhängig vom Monitorzustand) erfüllt ist.

Z.B. Puffer nicht voll: erst dann Ablegen eines Elements möglich,
oder Puffer nicht leer: erst dann Holen eines Elements möglich.

Beispiel (Carpark):

Simulation eines Parkhauses mit beschränkter Kapazität



“Arrivals“ darf nicht rotieren, wenn Parkhaus voll ist,
“Departures“ darf nicht rotieren, wenn Parkhaus leer ist.

5.2 Modellierung von Synchronisationsbedingungen

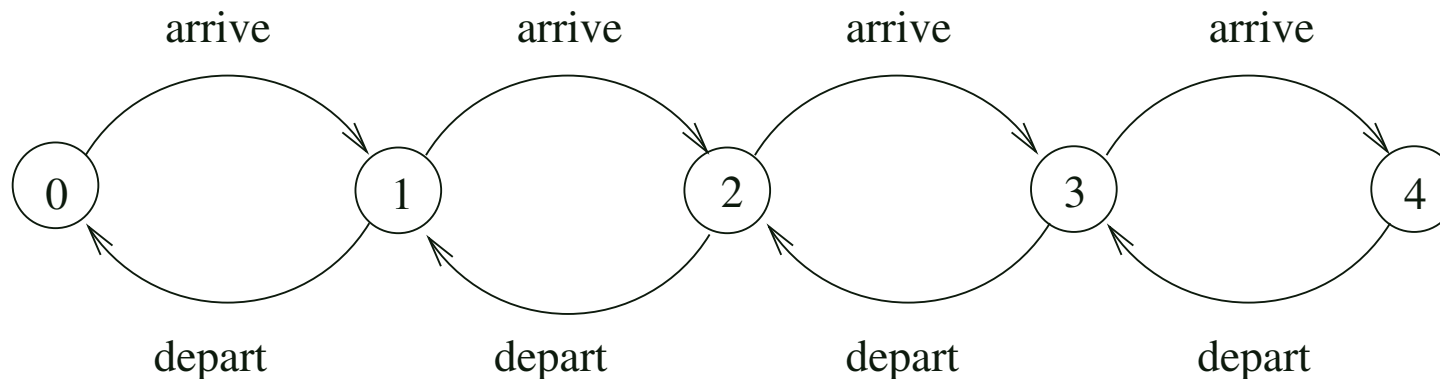
Beispiel (Modellierung des Carpark):

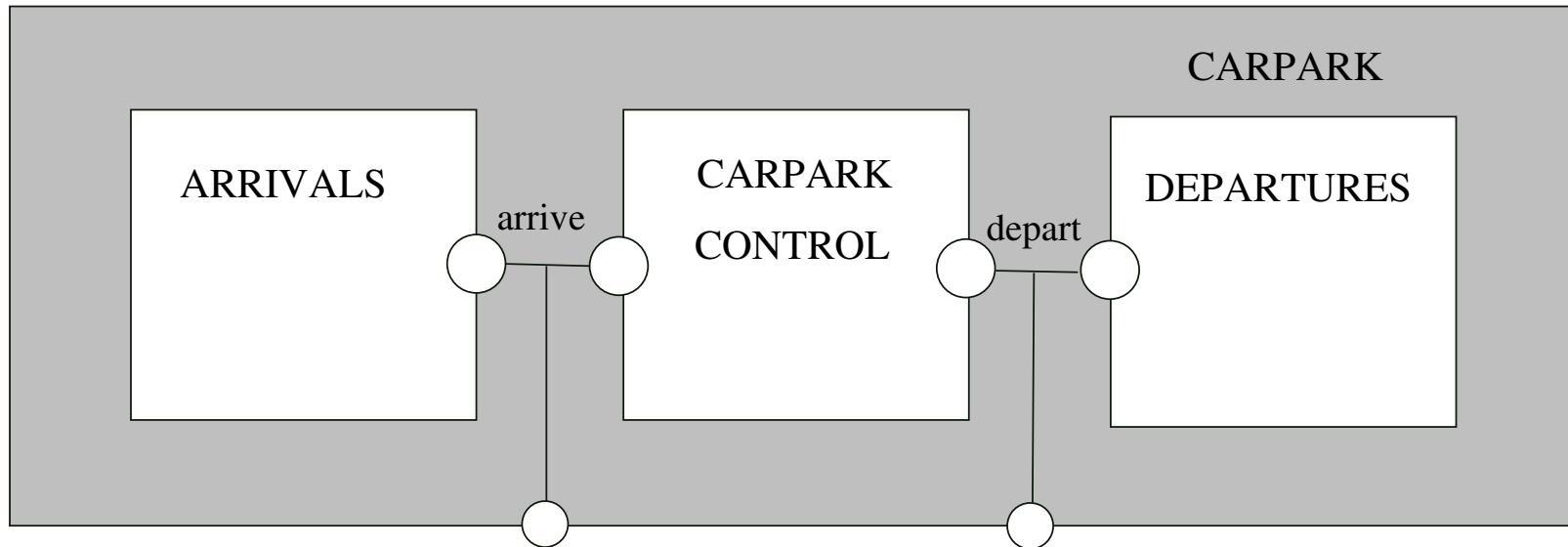
ARRIVALS = (arrive \rightarrow ARRIVALS).

DEPARTURES = (depart \rightarrow DEPARTURES).

CARPARKCONTROL(N=4) = SPACES[N],
 SPACES[i:0..N] = (when(i>0) arrive \rightarrow SPACES[i-1]
 | when(i<N) depart \rightarrow SPACES[i+1]).

||CARPARK = (ARRIVALS || CARPARKCONTROL(4) || DEPARTURES).





Strukturdiagramm des Carpark-Modells

Schema zur Modellierung von Synchronisationsbedingungen

$$P1 = (op1 \rightarrow P1),$$
$$P2 = (op2 \rightarrow P2).$$
$$\text{MONITOR} = \text{MON}[\text{init}],$$
$$\text{MON}[\text{state:T}] =$$
$$\quad (\text{when } (\text{cond1}) \text{ op1} \rightarrow \text{MON}[\text{nextState1}]$$
$$\quad | \text{when } (\text{cond2}) \text{ op2} \rightarrow \text{MON}[\text{nextState2}]).$$
$$\|\text{SYS} = (P1 \parallel P2 \parallel \text{MONITOR}).$$

Bemerkung:

- Die Synchronisationsbedingungen werden durch Wächter für Aktionen ausgedrückt.
- Die Bedingungen hängen vom Monitorzustand ab.
- Die Monitorzustände werden durch Prozessindizes (in lokalen Prozessen) modelliert.

Klassifizierung von Prozessen

- Im Carpark-Beispiel sind ARRIVALS und DEPARTURES *aktive* Komponenten. Sie initiieren die Aktionen “arrive“ und “depart“ (Output-Aktionen).
- CARPARKCONTROL ist eine *passive* Komponente, die auf Aktionen reagiert (Input-Aktionen).
- Aktive Komponenten werden durch Threads implementiert, passive Komponenten durch Monitore.

5.3 Implementierung von Synchronisationsbedingungen in Java

Zur Realisierung von Synchronisationsbedingungen verwenden wir die Methoden

```
public final void wait() throws InterruptedException  
public final void notify() bzw. notifyAll()
```

der allgemeinsten Klasse `Object`.

Wirkungsweise:

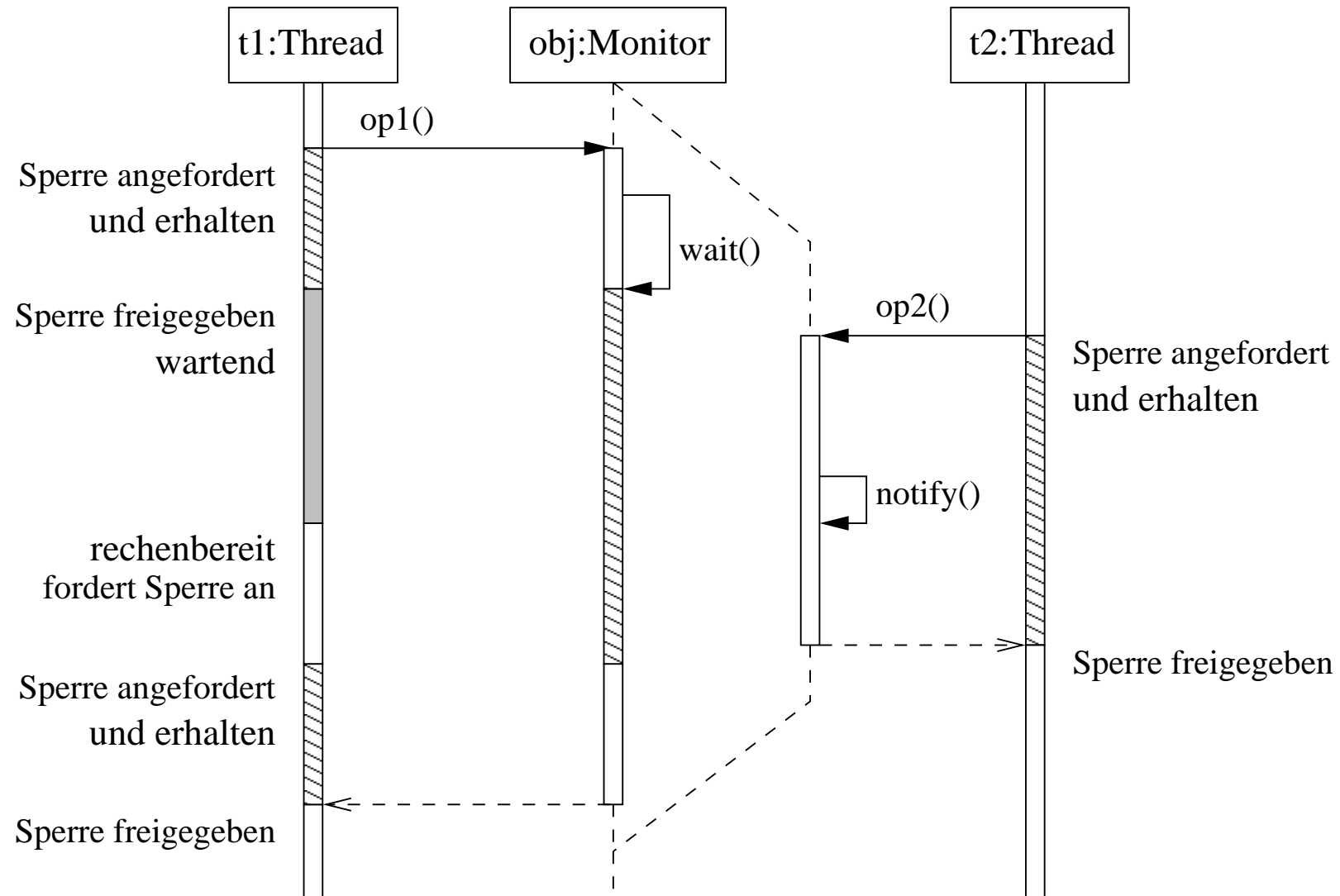
Jedes Objekt `obj` besitzt neben seiner Sperre eine Menge `w` von wartenden Threads. Die folgenden Operationen werden in einer unteilbaren Aktion ausgeführt:

- `obj.wait()`:
 - Der gerade ausgeführte Thread `t` gibt die Sperre von `obj` frei.
 - `t` wird in die Wartemenge `w` von `obj` eingereiht.
- `obj.notify()`:
 - Ein beliebiger Thread wird aus `w` freigelassen ("geweckt") und ist wieder rechenbereit.
- `obj.notifyAll()`:
 - Alle Threads werden aus der Wartemenge `w` von `obj` freigelassen.

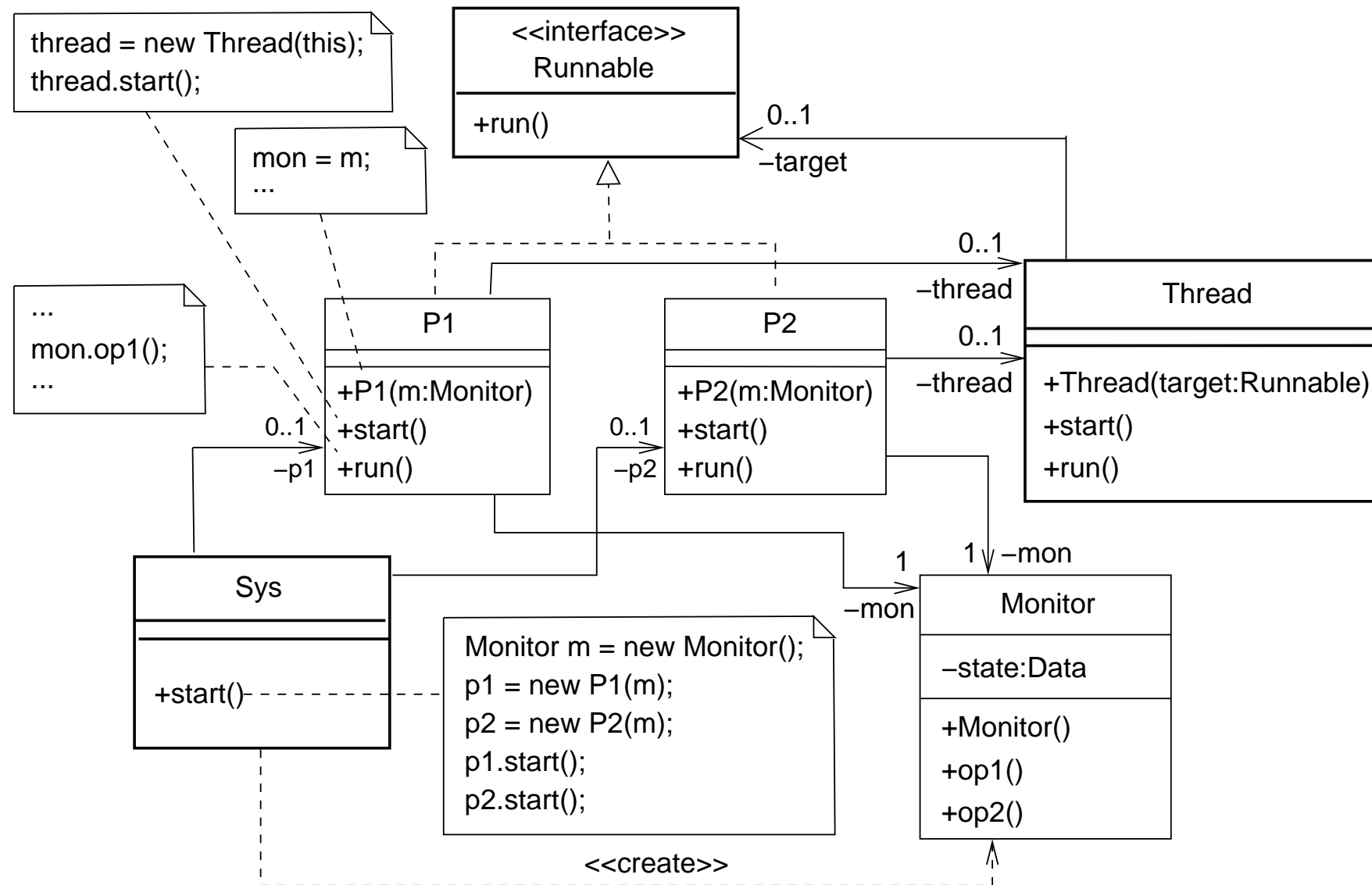
Beachte:

1. `wait` und `notify` (bzw. `notifyAll`) dürfen nur innerhalb von synchronisierten Blöcken aufgerufen werden.
2. Wird ein Thread `t` geweckt, dann muss er sich wieder um die Sperrre von `obj` bewerben. Die Sperre bleibt solange bei demjenigen Thread, innerhalb dessen “`notify`” aufgerufen wurde, bis dieser sie freigibt (“Signal and Continue“-Prinzip).
3. Sobald ein geweckter Thread `t` die Sperre von `obj` wieder erhält, fährt er dort fort, wo er aufgehört hat.

Sequenzdiagramm mit "wait" und "notify"



Schema zur Implementierung von Synchronisationsbedingungen



```
public class Monitor {
    private Data state;

    public synchronized void op1() throws InterruptedException {
        while (!cond1) wait();

        // modify monitor state

        notify(); // or notifyAll();
    }

    public synchronized void op2() throws InterruptedException {
        while (!cond2) wait();

        // modify monitor state

        notify(); // or notifyAll();
    }
}
```

Regel zur Implementierung von Synchronisationsbedingungen

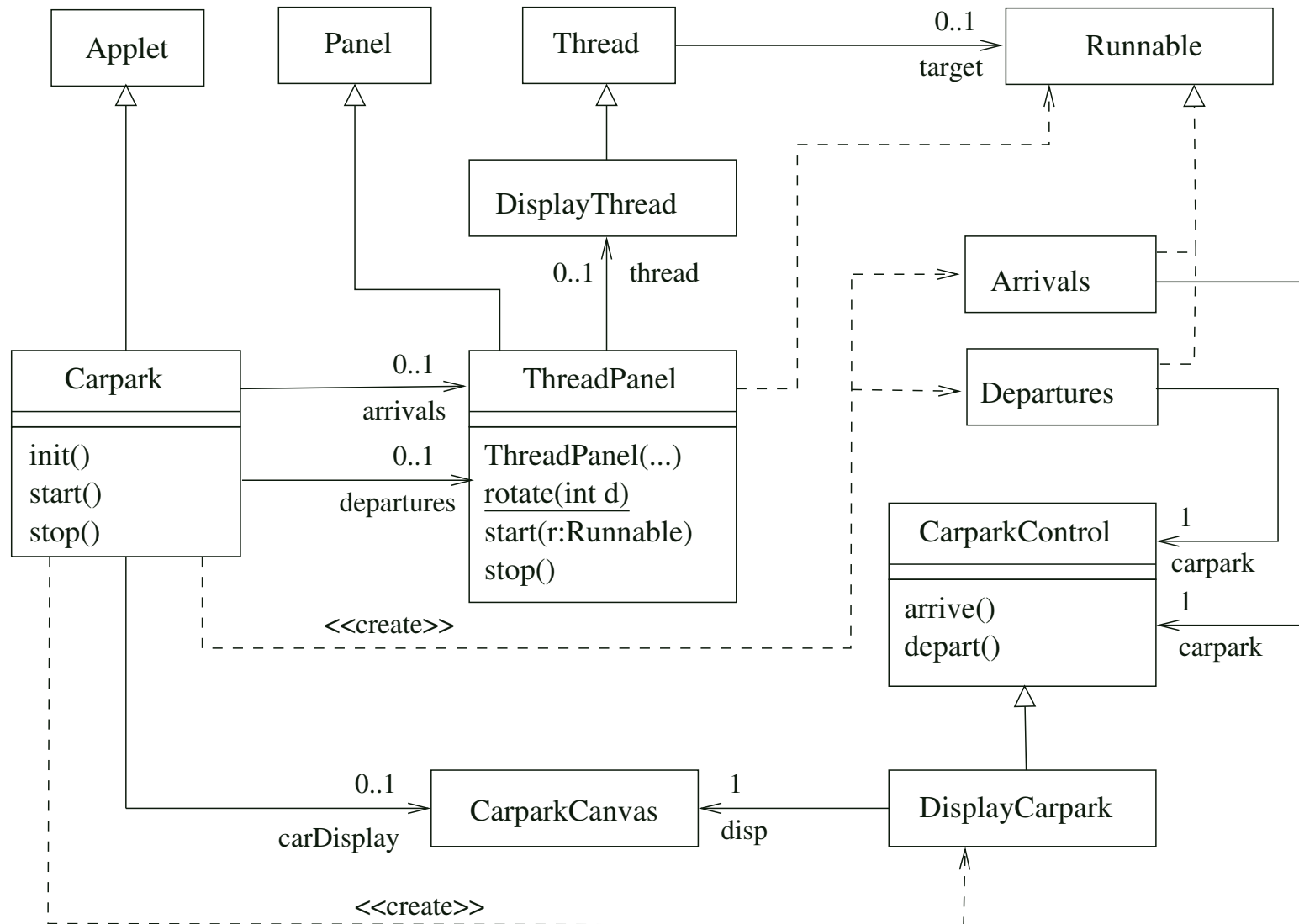
FSP: `when (cond) op → MONITOR[nextState]`

```
Java: public synchronized void op() throws InterruptedException {  
    while (!cond) wait();  
    ... // monitor state = nextState  
    notifyAll();  
}
```

Beachte:

Die `while`-Schleife ist notwendig, weil bei der Fortführung die Synchronisationsbedingung nicht notwendigerweise gelten muss.

Beispiel (Implementierung des Carpark):



Realisierung des FSP-Monitors CARPARKCONTROL durch die Klasse CarparkControl:

```
class CarparkControl {
    protected int capacity; // capacity=N in FSP
    protected int spaces; // spaces=i in FSP: freie Plaetze

    CarparkControl(int n) {
        capacity = spaces = n;
    }
    synchronized void arrive() throws InterruptedException {
        while (spaces<=0) wait();
        spaces--;
        notify(); // hier nur 1 anderer Prozess möglich
    }
    synchronized void depart() throws InterruptedException {
        while (spaces>=capacity) wait();
        spaces++;
        notify(); // hier nur 1 anderer Prozess möglich
    }
}
```


Realisierung der aktiven Komponenten ARRIVALS und DEPARTURES durch die Klassen Arrivals und Departures:

```
class Arrivals implements Runnable {
    private CarparkControl carpark;

    Arrivals(CarparkControl c) {
        carpark = c;
    }
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(330);
                carpark.arrive();
                ThreadPanel.rotate(30);
            }
        } catch (InterruptedException e) {}
    }
}
```

```
class Departures implements Runnable {
    private CarparkControl carpark;

    Departures(CarparkControl c) {
        carpark = c;
    }
    public void run() {
        try {
            while(true) {
                ThreadPanel.rotate(180);
                carpark.depart();
                ThreadPanel.rotate(180);
            }
        } catch (InterruptedException e) {}
    }
}
```

Realisierung des Applets Carpark:

```
class Carpark extends Applet {
    ThreadPanel arrivals;
    ThreadPanel departures;

    ...

    public void start() {
        CarparkControl c = new DisplayCarpark(carDisplay, places);
        arrivals.start(new Arrivals(c));
        departures.start(new Departures(c));
    }

    public void stop() {
        arrivals.stop();
        departures.stop();
    }
}
```

5.4 Semaphore

Das Semaphor-Konzept [Dijkstra 1968]

Ein Semaphor s besteht aus

- einer Integer-Variablen v zusammen mit
- einer Warteschlange w für die bezüglich des Semaphors blockierten Prozesse.

Es gibt genau zwei Operationen, die von einem Prozess P auf ein Semaphor s angewendet werden können:

down(s): if $v > 0$ then $v = v - 1$;
 else $w = w \cup \{P\}$

up(s): if $w = \emptyset$ then $v = v + 1$;
 else $w = w \setminus \{Q\}$ for some (previously blocked) process Q

Bemerkung

- Es wird vorausgesetzt, dass ein Semaphor mit einem Wert ≥ 0 (für v) initialisiert ist.
- Bei einem binären Semaphor nimmt v nur die Werte 0 und 1 an.

Modellierung von Semaphoren

const Max = 3

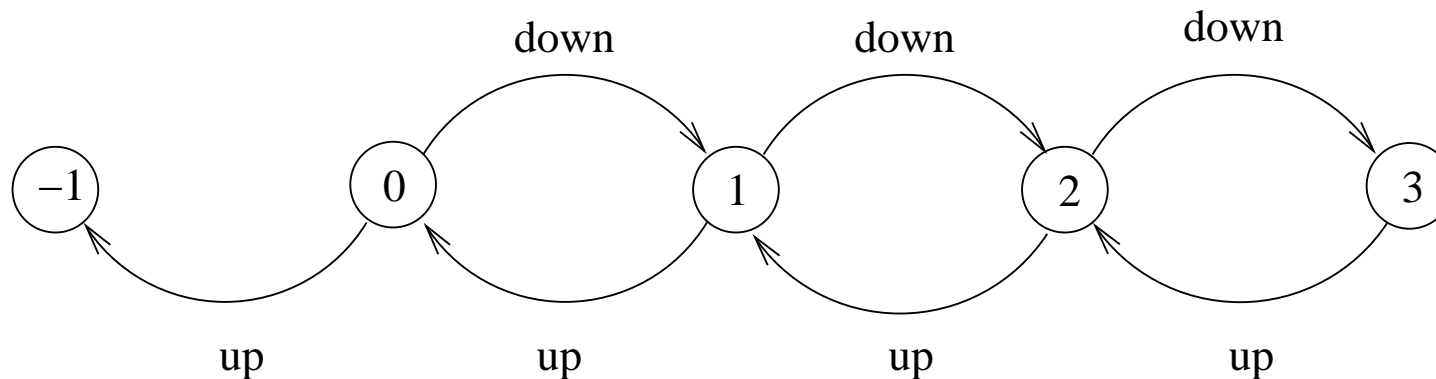
range Int = 0..Max

SEMAPHORE(N=3) = SEMA[N],

SEMA[v:Int] = (when (v > 0) down → SEMA[v-1]
 | up → SEMA[v+1]),

SEMA[Max+1] = ERROR.

Zugehöriges LTS:



Wechselseitiger Ausschluss mit Semaphoren

$A = (a.do \rightarrow a.down \rightarrow a.critical \rightarrow a.up \rightarrow A).$

$B = (b.do \rightarrow b.down \rightarrow b.critical \rightarrow b.up \rightarrow B).$

$\parallel \text{SYS} = (A \parallel B \parallel \{a,b\}::\text{SEMAPHORE}(1)).$

Simulation von Semaphoren in Java

```
class Semaphore {
    private int value;

    public Semaphore (int initial) {
        value = initial;
    }

    public synchronized void down() throws InterruptedException {
        while (value==0) wait();
        value--;
    }

    public synchronized void up() {
        value++;
        notify();
    }
}
```